Spring Social Reference Manual

Craig Walls
Keith Donald

Spring Social Reference Manual

by Craig Walls and Keith Donald

1.0.0.RELEASE

© SpringSource Inc., 2011

Table of Contents

1. Spring Social Overview

1.1 Introduction

The Spring Social project enables your applications to establish Connections with Software-as-a-Service (SaaS) Providers such as Facebook and Twitter to invoke APIs on behalf of Users.

1.2 Socializing applications

The phrase "social networking" often refers to efforts aimed at bringing people together. In the software world, those efforts take the form of online social networks such as Facebook, Twitter, and LinkedIn. Over half a billion of this world's internet users have flocked to these services to keep frequent contact with family, friends, and colleagues.

Under the surface, however, these services are just software applications that gather, store, and process information. Just like so many applications written before, these social networks have users who sign in and perform some activity offered by the service.

What makes these applications a little different than traditional applications is that the data that they collect represent some facet of their users' lives. What's more, these applications are more than willing to share that data with other applications, as long as the user gives permission to do so. This means that although these social networks are great at bringing people together, as software services they also excel at bringing applications together.

To illustrate, imagine that Paul is a member of an online movie club. A function of the movie club application is to recommend movies for its members to watch and to let its members maintain a list of movies that they have seen and those that they plan to see. When Paul sees a movie, he signs into the movie club site, checks the movie off of his viewing list, and indicates if he liked the movie or not. Based on his responses, the movie club application can tailor future recommendations for Paul to see.

On its own, the movie club provides great value to Paul, as it helps him choose movies to watch. But Paul is also a Facebook user. And many of Paul's Facebook friends also enjoy a good movie now and then. If Paul were able to connect his movie club account with his Facebook profile, the movie club application could offer him a richer experience. Perhaps when he sees a movie, the application could post a message on his Facebook wall indicating so. Or when offering suggestions, the movie club could factor in the movies that his Facebook friends liked.

Social integration is a three-way conversation between a service provider, a service consumer, and a user who holds an account on both the provider and consumer. All interactions between the consumer and the service provider are scoped to the context of the user's profile on the service provider.

In the narrative above, Facebook is the service provider, the movie club application is the service consumer, and Paul is the user of both. The movie club application may interact with Facebook on behalf of Paul, accessing whatever Facebook data and functionality that Paul permits, including retrieving Paul's friends and posting messages to his wall.

From the user's perspective, both applications provide some valuable functionality. But by connecting the user's account on the consumer application with his account on the provider application, the user brings together two applications that can now offer the user more value than they could individually.

With Spring Social, your application can play the part of the service consumer, interacting with a service provider on behalf of its users. The key features of Spring Social are:

- A "Connect Framework" that handles the core authorization and connection flow with service providers.
- A "Connect Controller" that handles the OAuth exchange between a service provider, consumer, and user in a web application environment.
- A "Signin Controller" that allows users to authenticate with your application by signing in with their Provider accounts, such as their Twitter or Facebook accounts.

In addition, there are a handful of provider-specific modules that extend Spring Social to enable integration with popular SaaS providers, including Facebook and Twitter.

1.3 How to get

The core Spring Social project consists of the modules described in Table 1.1, “Spring Social Modules”.

Table 1.1. Spring Social Modules

Name	Description
spring-social-core	Spring Social's Connect Framework and OAuth client support.
spring-social-web	Spring Social's `ConnectController` which uses the Connect Framework to manage connections in a web application environment.
spring-social-test	Support for testing Connect implementations and API bindings.

Which of these modules your application needs will largely depend on what facets of Spring Social you intend to use. At very minimum, you'll need the core module in your application's classpath:

```
<dependency>
    <groupId>org.springframework.social</groupId>
    <artifactId>spring-social-core</artifactId>
    <version>${spring-social.version}</version>
</dependency>
```

To let Spring Social handle the back-and-forth authorization handshake between your web application and a service provider, you'll need the web module:

```
<dependency>
    <groupId>org.springframework.social</groupId>
    <artifactId>spring-social-web</artifactId>
    <version>${spring-social.version}</version>
</dependency>
```

If you're developing your own client module (Chapter 3, *Adding Support for a New Service Provider*) and API binding, you'll need the test module to test it:

```
<dependency>
    <groupId>org.springframework.social</groupId>
    <artifactId>spring-social-test</artifactId>
    <version>${spring-social.version}</version>
</dependency>
```

If you are developing against a milestone or release candidate version, such as 1.0.0.M1 or 1.0.0.RC1, you will need to add the following repository in order to resolve the artifact:

```
<repository>
    <id>org.springframework.maven.milestone</id>
    <name>Spring Maven Milestone Repository</name>
    <url>http://maven.springframework.org/milestone</url>
</repository>
```

If you are testing out the latest nightly build version (e.g. 1.0.0.BUILD-SNAPSHOT), you will need to add the following repository:

```
<repository>
    <id>org.springframework.maven.snapshot</id>
    <name>Spring Maven Snapshot Repository</name>
    <url>http://maven.springframework.org/snapshot</url>
</repository>
```

Client modules

In addition to modules that make up the core Spring Social project, there are a number of provider-specific client modules that are released separately that provide connectivity and API bindings to popular SaaS providers. These client modules are listed in Table 1.2, “Spring Social Client Modules”.

Table 1.2. Spring Social Client Modules

Name	Maven group ID	Maven artifact ID
Spring Social Facebook [http://static.springsource.org/spring-social-facebook/docs/1.0.x/reference/html/]	org.springframework.social	spring-social-facebook
Spring Social Twitter [http://static.springsource.org/	org.springframework.social	spring-social-twitter

Name	Maven group ID	Maven artifact ID
spring-social-twitter/docs/1.0.x/reference/html/]		
Spring Social LinkedIn [http://static.springsource.org/spring-social-linkedin/docs/1.0.x/reference/html/]	org.springframework.social	spring-social-linkedin
Spring Social TripIt [http://static.springsource.org/spring-social-tripit/docs/1.0.x/reference/html/]	org.springframework.social	spring-social-tripit
Spring Social GitHub [http://static.springsource.org/spring-social-github/docs/1.0.x/reference/html/]	org.springframework.social	spring-social-github
Spring Social Gowalla [http://static.springsource.org/spring-social-gowalla/docs/1.0.x/reference/html/]	org.springframework.social	spring-social-gowalla

All of these modules are optional, depending on the connectivity needs of your application. For instance, if your application will connect with Facebook, you'll want to add the Facebook module to your project:

```
<dependency>
    <groupId>org.springframework.social</groupId>
    <artifactId>spring-social-facebook</artifactId>
    <version>${spring-social-facebook.version}</version>
</dependency>
```

Note that each of the client modules will progress and release on a different schedule than Spring Social. Consequently, the version numbers for any given client module may not align with Spring Social or any other client module.

Refer to each client module's reference documentation for details on connectivity and the API binding.

1.4 Dependencies

Spring Social depends on a few things to run. Most dependencies are optional and an effort has been made to keep the required dependencies to a minimum. The project dependencies are described in this section.

Java

Spring Social requires Java 1.5 or greater.

Java Servlet API

The Spring Social web support requires Java Servlet 2.5 or greater (Tomcat 6+).

Spring Framework

Spring Social depends on RestTemplate provided by the core Spring Framework [http://www.springsource.org/documentation] in the spring-web module. It requires Spring Framework version 3.0.5 or above. Spring Framework 3.1 is recommended to take advantage of several RestTemplate improvements.

If you are using Spring Social with Spring Framework 3.0.5 or >, make sure you explicitly add the spring-web dependency to your build:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>3.0.6.RELEASE</version>
</dependency>
```

Maven's dependency management favors "nearest" dependencies, so your project's definition of the spring-web dependency will override Spring Social's transitive dependency on the recommended 3.1 version.

Gradle, on the other hand, favors the newest dependency. If you're using Gradle to build your project, you'll need to also set the dependency's `force` property to `true` to force Gradle to resolve your chosen version of Spring:

```
dependencies {
    compile ("org.springframework:spring-web:3.0.6.RELEASE") { force=true }
}
```

Spring Security Crypto

If you're not already using Spring Security to secure your application, you'll need to add the standalone crypto module. This is required for OAuth1 request signing and encrypting credentials when persisting Connection data. If you're already using Spring Security, there is nothing for you to do because the crypto library comes included.

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-crypto</artifactId>
    <version>3.1.0.RC3</version>
</dependency>
```

Apache HttpComponents

Spring Social has an optional dependency on Apache HttpComponents [http://hc.apache.org/httpcomponents-client-ga]. If the HttpComponents HttpClient library is present, it will use it as the HTTP client (which is generally recommended). Otherwise, it will fall back on standard J2SE facilities.

```
<dependency>
    <groupId>org.apache.httpcomponents</groupId>
    <artifactId>httpclient</artifactId>
    <version>4.1.2</version>
</dependency>
```

Although shown here to depend on version 4.1.1 of the HttpClient library, Spring Social can also with with 4.0.X versions of HttpClient.

Jackson JSON Processor

Spring Social's provider API bindings rely on the Jackson JSON Processor [http://jackson.codehaus.org/] to map JSON responses to Java objects. Each binding, such as Facebook or Twitter, transitively depends on Jackson 1.8.5, so there's nothing special to do to add Jackson to your project's Maven or Gradle build.

1.5 Sample Code

We have created a few sample applications to illustrate the capabilities of Spring Social. To obtain the https://github.com/SpringSource/spring-social-samples code, use the following git command:

```
git clone git://github.com/SpringSource/spring-social-samples.git
```

The Spring Social Samples project includes the following samples:

- spring-social-quickstart - Designed to get you up and running quickly.
- spring-social-quickstart-30x - Designed to get you up and running quickly as well as using Spring Social with Spring 3.0.x.
- spring-social-showcase - Illustrates most of Spring Social's features.
- spring-social-movies - Shows how to extend Spring Social to implement a new ServiceProvider and API binding.
- spring-social-twitter4j - Shows how to extend Spring Social and re-use an existing API binding.

- spring-social-popup - Shows how to use Spring Social to drive a browser popup-based connection flow.
- spring-social-canvas - Demonstrates how to use Spring Social within a Facebook Canvas application.

2. Service Provider 'Connect' Framework

The `spring-social-core` module includes a *Service Provider 'Connect' Framework* for managing connections to Software-as-a-Service (SaaS) providers such as Facebook and Twitter. This framework allows your application to establish connections between local user accounts and accounts those users have with external service providers. Once a connection is established, it can be be used to obtain a strongly-typed Java binding to the ServiceProvider's API, giving your application the ability to invoke the API on behalf of a user.

To illustrate, consider Facebook as an example ServiceProvider. Suppose your application, AcmeApp, allows users to share content with their Facebook friends. To support this, a connection needs to be established between a user's AcmeApp account and her Facebook account. Once established, a Facebook instance can be obtained and used to post content to the user's wall. Spring Social's 'Connect' framework provides a clean API for managing service provider connections such as this.

2.1 Core API

The `Connection<A>` interface models a connection to an external service provider such as Facebook:

```
public interface Connection<A> {

    ConnectionKey getKey();

    String getDisplayName();

    String getProfileUrl();

    String getImageUrl();

    void sync();

    boolean test();

    boolean hasExpired();

    void refresh();

    UserProfile fetchUserProfile();

    void updateStatus(String message);

    A getApi();

    ConnectionData createData();

}
```

Each connection is uniquely identified by a composite key consisting of a providerId (e.g. 'facebook') and connected providerUserId (e.g. '1255689239', for Keith Donald's Facebook ID). This key tells you what provider user the connection is connected to.

A connection has a number of meta-properties that can be used to render it on a screen, including a displayName, profileUrl, and imageUrl. As an example, the following HTML template snippet could be used to generate a link to the connected user's profile on the provider's site:

```
<img src="${connection.imageUrl}" /> <a href="${connection.profileUrl}">${connection.displayName}</a>
```

The value of these properties may depend on the state of the provider user's profile. In this case, sync() can be called to synchronize these values if the user's profile is updated.

A connection can be tested to determine if its authorization credentials are valid. If invalid, the connection may have expired or been revoked by the provider. If the connection has expired, a connection may be refreshed to renew its authorization credentials.

A connection provides several operations that allow the client application to invoke the ServiceProvider's API in a uniform way. This includes the ability to fetch a model of the user's profile and update the user's status in the provider's system.

A connection's parameterized type <A> represents the Java binding to the ServiceProvider's native API. An instance of this API binding can be obtained by calling `getApi()`. As an example, a Facebook connection instance would be parameterized as Connection<Facebook>. `getApi()` would return a Facebook instance that provides a Java binding to Facebook's graph API for a specific Facebook user.

Finally, the internal state of a connection can be captured for transfer between layers of your application by calling `createData()`. This could be used to persist the connection in a database, or serialize it over the network.

To put this model into action, suppose we have a reference to a Connection<Twitter> instance. Suppose the connected user is the Twitter user with screen name 'kdonald'.

1. Connection#getKey() would return ('twitter', '14718006') where '14718006' is @kdonald's Twitter-assigned user id that never changes.
2. Connection#getDisplayName() would return '@kdonald'.
3. Connection#getProfileUrl() would return 'http://twitter.com/kdonald'.
4. Connection#getImageUrl() would return 'http://a0.twimg.com/profile_images/105951287/IMG_5863_2_normal.jpg'.
5. Connection#sync() would synchronize the state of the connection with @kdonald's profile.
6. Connection#test() would return true indicating the authorization credentials associated with the Twitter connection are valid. This assumes Twitter has not revoked the AcmeApp client application, and @kdonald has not reset his authorization credentials (Twitter connections do not expire).
7. Connection#hasExpired() would return false.
8. Connection#refresh() would not do anything since connections to Twitter do not expire.
9. Connection#fetchUserProfile() would make a remote API call to Twitter to get @kdonald's profile data and normalize it into a UserProfile model.

10.Connection#updateStatus(String) would post a status update to @kdonald's timeline.

11.Connection#getApi() would return a Twitter giving the client application access to the full capabilities of Twitter's native API.

12.Connection#createData() would return ConnectionData that could be serialized and used to restore the connection at a later time.

2.2 Establishing connections

So far we have discussed how existing connections are modeled, but we have not yet discussed how new connections are established. The manner in which connections between local users and provider users are established varies based on the authorization protocol used by the ServiceProvider. Some service providers use OAuth, others use Basic Auth, others may use something else. Spring Social currently provides native support for OAuth-based service providers, including support for OAuth 1 and OAuth 2. This covers the leading social networks, such as Facebook and Twitter, all of which use OAuth to secure their APIs. Support for other authorization protocols can be added by extending the framework.

Each authorization protocol is treated as an implementation detail where protocol-specifics are kept out of the core Connection API. A ConnectionFactory abstraction encapsulates the construction of connections that use a specific authorization protocol. In the following sections, we will discuss the major ConnectionFactory classes provided by the framework. Each section will also describe the protocol-specific flow required to establish a new connection.

OAuth2 service providers

OAuth 2 is rapidly becoming a preferred authorization protocol, and is used by major service providers such as Facebook, Github, Foursquare, Gowalla, and 37signals. In Spring Social, a OAuth2ConnectionFactory is used to establish connections with a OAuth2-based service provider:

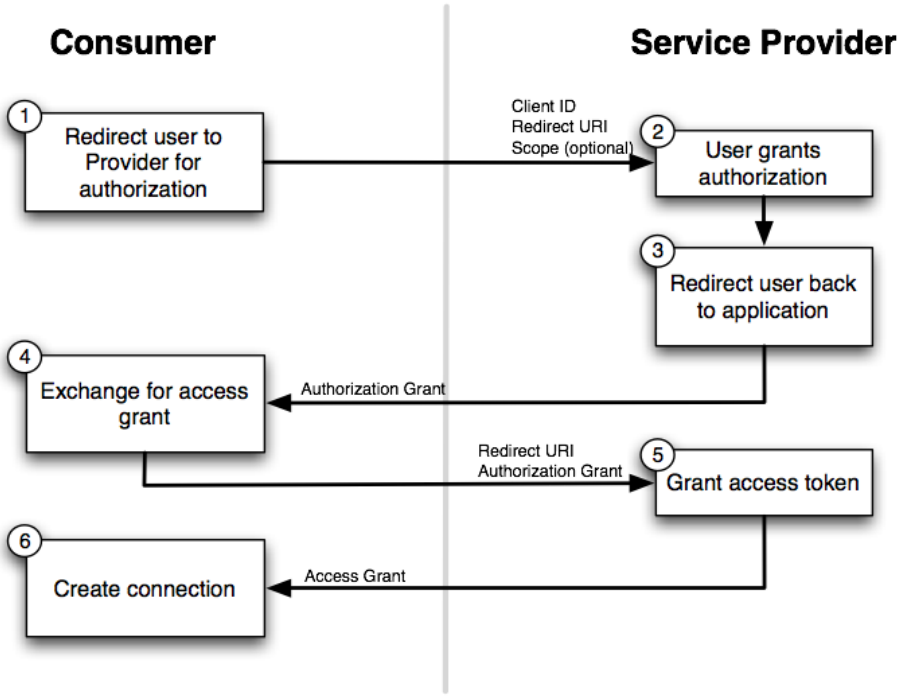
```
public class OAuth2ConnectionFactory<A> extends ConnectionFactory<A> {

    public OAuth2Operations getOAuthOperations();

    public Connection<A> createConnection(AccessGrant accessGrant);

    public Connection<A> createConnection(ConnectionData data);

}
```

`getOAuthOperations()` returns an API to use to conduct the authorization flow, or "OAuth Dance", with a service provider. The result of this flow is an `AccessGrant` that can be used to establish a connection with a local user account by calling `createConnection`. The OAuth2Operations interface is shown below:

```
public interface OAuth2Operations {

    String buildAuthorizeUrl(GrantType grantType, OAuth2Parameters parameters);

    String buildAuthenticateUrl(GrantType grantType, OAuth2Parameters parameters);
```

```
    AccessGrant exchangeForAccess(String authorizationCode, String redirectUri,
        MultiValueMap<String, String> additionalParameters);

    AccessGrant refreshAccess(String refreshToken, String scope,
            MultiValueMap<String, String> additionalParameters);

}
```

Callers are first expected to call buildAuthorizeUrl(GrantType, OAuth2Parameters) to construct the URL to redirect the user to for connection authorization. Upon user authorization, the authorizationCode returned by the provider should be exchanged for an AccessGrant. The AccessGrant should then used to create a connection. This flow is illustrated below:



As you can see, there is a back-and-forth conversation that takes place between the application and the service provider to grant the application access to the provider account. This exchange, commonly known as the "OAuth Dance", follows these steps:

1. The flow starts by the application redirecting the user to the provider's authorization URL. Here the provider displays a web page asking the user if he or she wishes to grant the application access to read and update their data.

2. The user agrees to grant the application access.

3. The service provider redirects the user back to the application (via the redirect URI), passing an authorization code as a parameter.

4. The application exchanges the authorization code for an access grant.

5. The service provider issues the access grant to the application. The grant includes an access token and a refresh token. One receipt of these tokens, the "OAuth dance" is complete.

6. The application uses the AccessGrant to establish a connection between the local user account and the external provider account. With the connection established, the application can now obtain a reference to the Service API and invoke the provider on behalf of the user.

The example code below shows use of a FacebookConnectionFactory to create a connection to Facebook using the OAuth2 server-side flow illustrated above. Here, FacebookConnectionFactory is a subclass of OAuth2ConnectionFactory:

```
FacebookConnectionFactory connectionFactory =
    new FacebookConnectionFactory("clientId", "clientSecret");
OAuth2Operations oauthOperations = connectionFactory.getOAuthOperations();
OAuth2Parameters params = new OAuth2Parameters();
params.setRedirectUri("https://my-callback-url");
String authorizeUrl = oauthOperations.buildAuthorizeUrl(GrantType.AUTHORIZATION_CODE, params);
response.sendRedirect(authorizeUrl);

// upon receiving the callback from the provider:
AccessGrant accessGrant = oauthOperations.exchangeForAccess(authorizationCode, "https://my-callback-url", null)
Connection<Facebook> connection = connectionFactory.createConnection(accessGrant);
```

The following example illustrates the client-side "implicit" authorization flow also supported by OAuth2. The difference between this flow and the server-side "authorization code" flow above is the provider callback directly contains the access grant (no additional exchange is necessary). This flow is appropriate for clients incapable of keeping the access grant credentials confidential, such as a mobile device or JavaScript-based user agent.

```
FacebookConnectionFactory connectionFactory =
    new FacebookConnectionFactory("clientId", "clientSecret");
OAuth2Operations oauthOperations = connectionFactory.getOAuthOperations();
OAuth2Parameters params = new OAuth2Parameters();
params.setRedirectUri("https://my-callback-url");
String authorizeUrl = oauthOperations.buildAuthorizeUrl(GrantType.IMPLICIT_GRANT, params);
response.sendRedirect(authorizeUrl);

// upon receiving the callback from the provider:
AccessGrant accessGrant = new AccessGrant(accessToken);
Connection<Facebook> connection = connectionFactory.createConnection(accessGrant);
```

OAuth1 service providers

OAuth 1 is the previous version of the OAuth protocol. It is more complex OAuth 2, and sufficiently different that it is supported separately. Twitter, Linked In, and TripIt are some of the well-known ServiceProviders that use OAuth 1. In Spring Social, the OAuth1ConnectionFactory allows you to create connections to a OAuth1-based Service Provider:

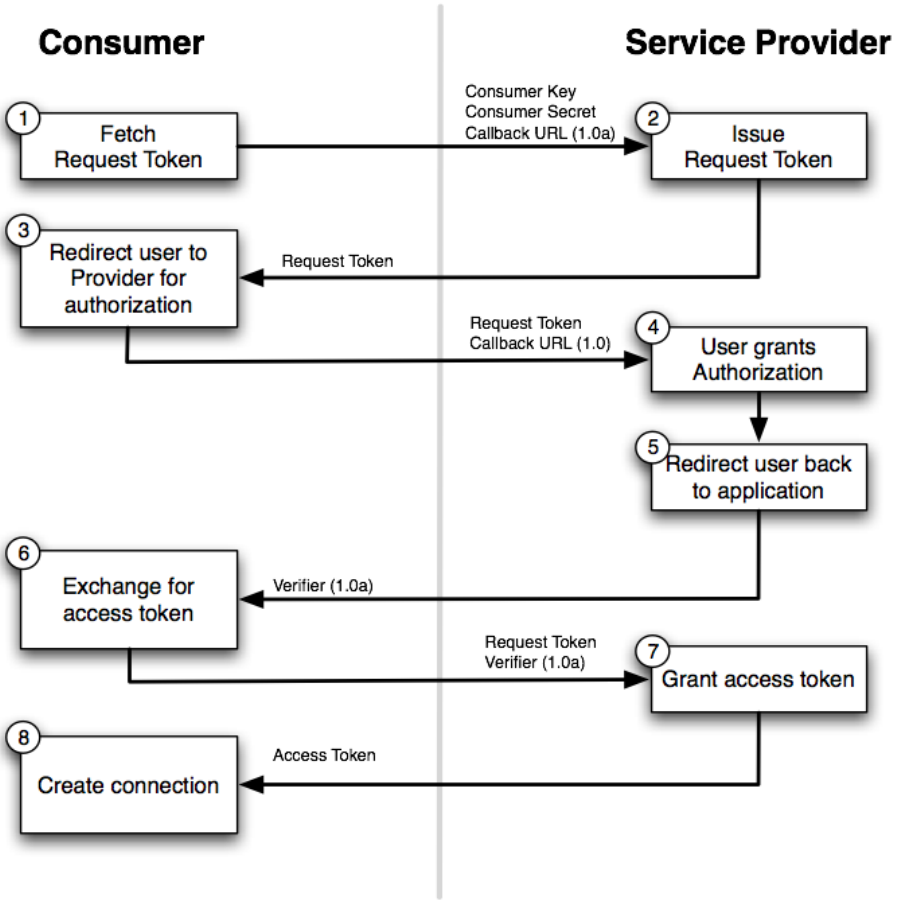
```
public class OAuth1ConnectionFactory<A> extends ConnectionFactory<A> {

    public OAuth1Operations getOAuthOperations();

    public Connection<A> createConnection(OAuthToken accessToken);
```

```
    public Connection<A> createConnection(ConnectionData data);

}
```

Like a OAuth2-based provider, `getOAuthOperations()` returns an API to use to conduct the authorization flow, or "OAuth Dance". The result of the OAuth 1 flow is an `OAuthToken` that can be used to establish a connection with a local user account by calling `createConnection`. The OAuth1Operations interface is shown below:

```
public interface OAuth1Operations {

    OAuthToken fetchRequestToken(String callbackUrl,
        MultiValueMap<String, String> additionalParameters);

    String buildAuthorizeUrl(String requestToken, OAuth1Parameters parameters);

    String buildAuthenticateUrl(String requestToken, OAuth1Parameters parameters);

    OAuthToken exchangeForAccessToken(AuthorizedRequestToken requestToken,
        MultiValueMap<String, String> additionalParameters);

}
```

Callers are first expected to call fetchNewRequestToken(String) to obtain a temporary token from the ServiceProvider to use during the authorization session. Next, callers should call buildAuthorizeUrl(String, OAuth1Parameters) to construct the URL to redirect the user to for connection authorization. Upon user authorization, the authorized request token returned by the provider should be exchanged for an access token. The access token should then used to create a connection. This flow is illustrated below:



1. The flow starts with the application asking for a request token. The purpose of the request token is to obtain user approval and it can only be used to obtain an access token. In OAuth 1.0a, the consumer callback URL is passed to the provider when asking for a request token.

2. The service provider issues a request token to the consumer.

3. The application redirects the user to the provider's authorization page, passing the request token as a parameter. In OAuth 1.0, the callback URL is also passed as a parameter in this step.

4. The service provider prompts the user to authorize the consumer application and the user agrees.

5. The service provider redirects the user's browser back to the application (via the callback URL). In OAuth 1.0a, this redirect includes a verifier code as a parameter. At this point, the request token is authorized.

6. The application exchanges the authorized request token (including the verifier in OAuth 1.0a) for an access token.

7. The service provider issues an access token to the consumer. The "dance" is now complete.

8. The application uses the access token to establish a connection between the local user account and the external provider account. With the connection established, the application can now obtain a reference to the Service API and invoke the provider on behalf of the user.

The example code below shows use of a TwitterConnectionFactory to create a connection to Facebook using the OAuth1 server-side flow illustrated above. Here, TwitterConnectionFactory is a subclass of OAuth1ConnectionFactory:

```
TwitterConnectionFactory connectionFactory =
    new TwitterConnectionFactory("consumerKey", "consumerSecret");
OAuth1Operations oauthOperations = connectionFactory.getOAuthOperations();
OAuthToken requestToken = oauthOperations.fetchRequestToken("https://my-callback-url", null);
String authorizeUrl = oauthOperations.buildAuthorizeUrl(requestToken, OAuth1Parameters.NONE);
response.sendRedirect(authorizeUrl);

// upon receiving the callback from the provider:
OAuthToken accessToken = oauthOperations.exchangeForAccessToken(
    new AuthorizedRequestToken(requestToken, oauthVerifier), null);
Connection<Twitter> connection = connectionFactory.createConnection(accessToken);
```

Registering ConnectionFactory instances

As you will see in subsequent sections of this reference guide, Spring Social provides infrastructure for establishing connections to one or more providers in a dynamic, self-service manner. For example, one client application may allow users to connect to Facebook, Twitter, and LinkedIn. Another might integrate Github and Pivotal Tracker. To make the set of connectable providers easy to manage and locate, Spring Social provides a registry for centralizing connection factory instances:

```
ConnectionFactoryRegistry registry = new ConnectionFactoryRegistry();
registry.addConnectionFactory(new FacebookConnectionFactory("clientId", "clientSecret"));
registry.addConnectionFactory(new TwitterConnectionFactory("consumerKey", "consumerSecret"));
registry.addConnectionFactory(new LinkedInConnectionFactory("consumerKey", "consumerSecret"));
```

This registry implements a locator interface that other objects can use to lookup connection factories dynamically:

```
public interface ConnectionFactoryLocator {

    ConnectionFactory<?> getConnectionFactory(String providerId);

    <A> ConnectionFactory<A> getConnectionFactory(Class<A> apiType);

    Set<String> registeredProviderIds();

}
```

Example usage of a ConnectionFactoryLocator is shown below:

```
// generic lookup by providerId
ConnectionFactory<?> connectionFactory = locator.getConnectionFactory("facebook");

// typed lookup by service api type
ConnectionFactory<Facebook> connectionFactory = locator.getConnectionFactory(Facebook.class);
```

2.3 Persisting connections

After a connection has been established, you may wish to persist it for later use. This makes things convenient for the user since a connection can simply be restored from its persistent form and does not need to be established again. Spring Social provides a ConnectionRepository interface for managing the persistence of a user's connections:

```
public interface ConnectionRepository {

    MultiValueMap<String, Connection<?>> findAllConnections();

    List<Connection<?>> findConnections(String providerId);

    <A> List<Connection<A>> findConnections(Class<A> apiType);

    MultiValueMap<String, Connection<?>> findConnectionsToUsers(
        MultiValueMap<String, String> providerUserIds);

    Connection<?> getConnection(ConnectionKey connectionKey);

    <A> Connection<A> getConnection(Class<A> apiType, String providerUserId);

    <A> Connection<A> getPrimaryConnection(Class<A> apiType);

    <A> Connection<A> findPrimaryConnection(Class<A> apiType);

    void addConnection(Connection<?> connection);

    void updateConnection(Connection<?> connection);

    void removeConnections(String providerId);

    void removeConnection(ConnectionKey connectionKey);

}
```

As you can see, this interface provides a number of operations for adding, updating, removing, and finding Connections. Consult the JavaDoc API of this interface for a full description of these operations. Note that all operations on this repository are scoped relative to the "current user" that has authenticated with your local application. For standalone, desktop, or mobile environments that only have one user this distinction isn't important. In a multi-user web application environment, this implies ConnectionRepository instances will be request-scoped.

For multi-user environments, Spring Social provides a UsersConnectionRepository that provides access to the global store of connections across all users:

```
public interface UsersConnectionRepository {

    String findUserIdWithConnection(Connection<?> connection);

    Set<String> findUserIdsConnectedTo(String providerId, Set<String> providerUserIds);
```

```
    ConnectionRepository createConnectionRepository(String userId);

}
```

As you can see, this repository acts as a factory for ConnectionRepository instances scoped to a single user, as well as exposes a number of multi-user operations. These operations include the ability to lookup the local userIds associated with connections to support provider user sign-in and "registered friends" scenarios. Consult the JavaDoc API of this interface for a full description.

JDBC-based persistence

Spring Social provides a JdbcUsersConnectionRepository implementation capable of persisting connections to a RDBMS. The database schema designed to back this repository is defined as follows:

```
create table UserConnection (userId varchar(255) not null,
    providerId varchar(255) not null,
    providerUserId varchar(255),
    rank int not null,
    displayName varchar(255),
    profileUrl varchar(512),
    imageUrl varchar(512),
    accessToken varchar(255) not null,
    secret varchar(255),
    refreshToken varchar(255),
    expireTime bigint,
    primary key (userId, providerId, providerUserId));
create unique index UserConnectionRank on UserConnection(userId, providerId, rank);
```

For convenience is bootstrapping the schema from a running application, this schema definition is available in the `spring-social-core` module as a resource at the path /org/springframework/social/connect/jdbc/JdbcUsersConnectionRepository.sql.

The implementation also provides support for encrypting authorization credentials so they are not stored in plain-text.

The example code below demonstrates construction and usage of a JdbcUsersConnectionRepository:

```
// JDBC DataSource pointing to the DB where connection data is stored
DataSource dataSource = ...;

// locator for factories needed to construct Connections when restoring from persistent form
ConnectionFactoryLocator connectionFactoryLocator = ...;

// encryptor of connection authorization credentials
TextEncryptor encryptor = ...;

UsersConnectionRepository usersConnectionRepository =
    new JdbcUsersConnectionRepository(dataSource, connectionFactoryLocator, encryptor);

// create a connection repository for the single-user 'kdonald'
ConnectionRepository repository = usersConnectionRepository.createConnectionRepository("kdonald");
```

```
// find kdonald's primary Facebook connection
Connection<Facebook> connection = repository.findPrimaryConnection(Facebook.class);
```

3. Adding Support for a New Service Provider

Spring Social makes it easy to add support for service providers that are not already supported by the framework. If you review the existing client modules, such as spring-social-twitter and spring-social-facebook, you will discover they are implemented in a consistent manner and they apply a set of well-defined extension points. In this chapter, you will learn how to add support for new service providers you wish to integrate into your applications.

3.1 Process overview

The process of adding support for a new service provider consists of several steps:

1. Create a source project for the client code e.g. `spring-social-twitter`.
2. Develop or integrate a Java binding to the provider's API e.g. `Twitter`.
3. Create a ServiceProvider model that allows users to authorize with the remote provider and obtain authorized API instances e.g. `TwitterServiceProvider`.
4. Create an ApiAdapter that maps the provider's native API onto the uniform Connection model e.g. `TwitterAdapter`.
5. Finally, create a ConnectionFactory that wraps the other artifacts up and provides a simple interface for establishing connections e.g. `TwitterConnectionFactory`.

The following sections of this chapter walk you through each of the steps with examples.

3.2 Creating a source project for the provider client code

A Spring Social client module is a standard Java project that builds a single jar artifact e.g. spring-social-twitter.jar. We recommend the code structure of a client module follow the guidelines described below.

Code structure guidelines

We recommend the code for a new Spring Social client module reside within the `org.springframework.social.{providerId}` base package, where {providerId} is a unique identifier you assign to the service provider you are adding support for. Consider some of the providers already supported by the framework as examples:

Table 3.1. Spring Social Client Modules

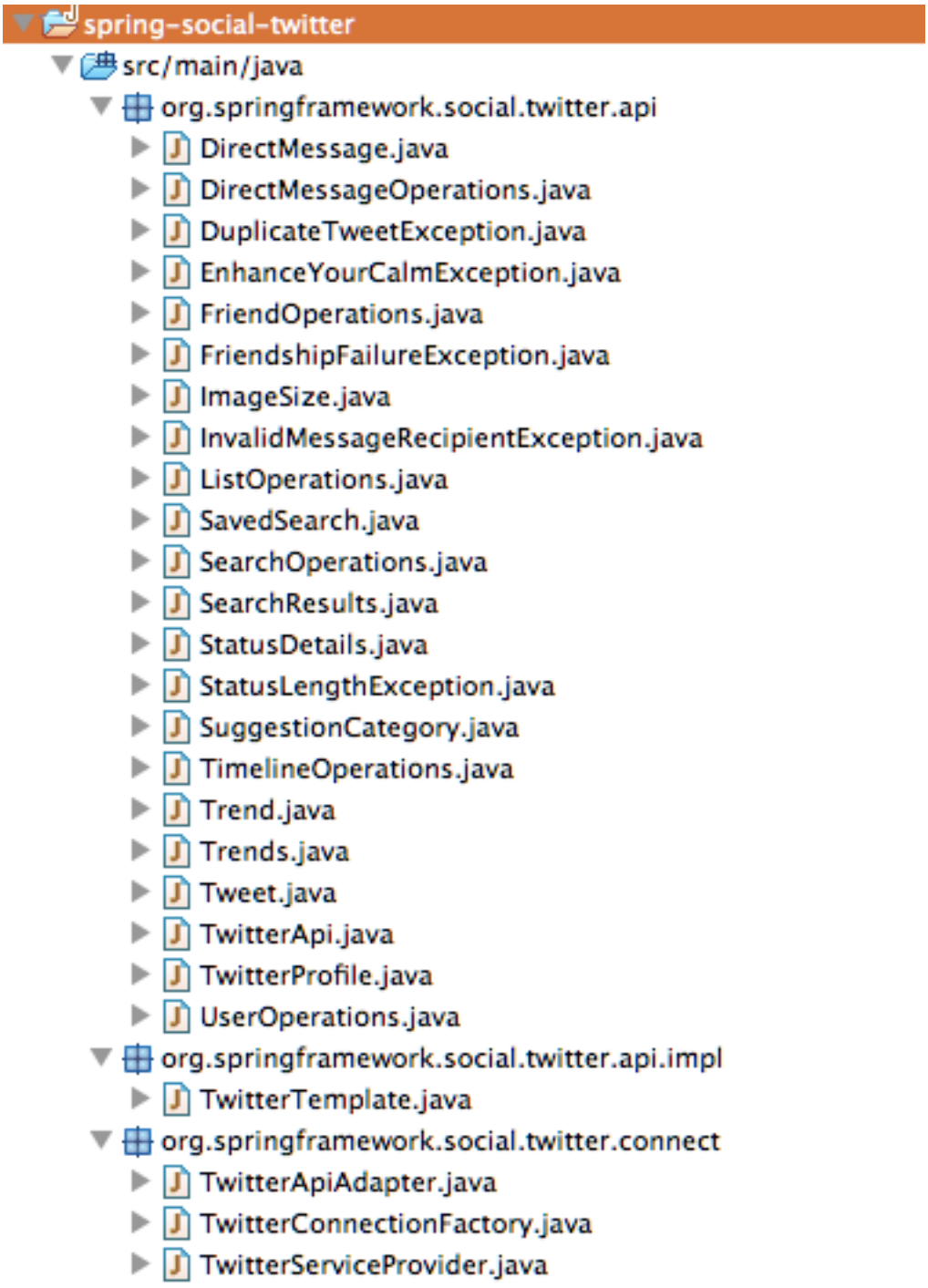
Provider ID	Artifact Name	Base Package
facebook	spring-social-facebook	org.springframework.social.facebook
twitter	spring-social-twitter	org.springframework.social.twitter

Within the base package, we recommend the following subpackage structure:

Table 3.2. Module Structure

Subpackage	Description
api	The public interface that defines the API binding.
api.impl	The implementation of the API binding.
connect	The types necessary to establish connections to the service provider.

You can see this recommended structure in action by reviewing one of the other client modules such as spring-social-twitter:



Here, the central service API type, Twitter, is located in the api package along with its supporting operations types and data transfer object types. The primary implementation of that interface, TwitterTemplate, is located in the api.impl package (along with other package-private impl types have that been excluded from this view). Finally, the connect package contains the implementations of various connect SPIs that enable connections to Twitter to be established and persisted.

3.3 Developing a Java binding to the provider's API

Spring Social favors the development of strongly-typed Java bindings to external service provider APIs. This provides a simple, domain-oriented interface for Java applications to use to consume the API. When adding support for a new service provider, if no suitable Java binding already exists you'll need to develop one. If one already exists, such as Twitter4j for example, it is possible to integrate it into the framework.

Designing a new Java API binding

API developers retain full control over the design and implementation of their Java bindings. That said, we offer several design guidelines in an effort to improve overall consistency and quality:

- *Favor separating the API binding interface from the implementation.* This is illustrated in the spring-social-twitter example in the previous section. There, "Twitter" is the central API binding type and it is declared in the org.springframework.social.twitter.api package with other public types. "TwitterTemplate" is the primary implementation of this interface and is located in the org.springframework.social.twitter.api.impl subpackage along with other package-private implementation types.
- *Favor organizing the API binding hierarchically by RESTful resource.* REST-based APIs typically expose access to a number of resources in an hierarchical manner. For example, Twitter's API provides access to "status timelines", "searches", "lists", "direct messages", "friends", "geo location", and "users". Rather than add all operations across these resources to a single flat "Twitter" interface, the Twitter interface is organized hierarchically:

```
public interface Twitter extends ApiBinding {

    DirectMessageOperations directMessageOperations();

    FriendOperations friendOperations();

    GeoOperations geoOperations();

    ListOperations listOperations();

    SearchOperations searchOperations();

    TimelineOperations timelineOperations();

    UserOperations userOperations();

}
```

DirectMessageOperations, for example, contains API bindings to Twitter's "direct_messages" resource:

```
public interface DirectMessageOperations {

    List<DirectMessage> getDirectMessagesReceived();

    List<DirectMessage> getDirectMessagesSent();

    void sendDirectMessage(String toScreenName, String text);

    void sendDirectMessage(long toUserId, String text);

    void deleteDirectMessage(long messageId);
}
```

Implementing a new Java API binding

API developers are free to implement their Java API binding with whatever REST/HTTP client they see fit. That said, Spring Social's existing API bindings such as spring-social-twitter all use Spring Framework's RestTemplate in conjunction with the Jackson JSON ObjectMapper and Apache HttpComponents HTTP client. RestTemplate is a popular REST client that provides a uniform object mapping interface across a variety of data exchange formats (JSON, XML, etc). Jackson is the leading Java-based JSON marshalling technology. Apache HttpComponents has proven to be the most robust HTTP client (if it is not available on the classpath Spring Social will fallback to standard J2SE facilities, however). To help promote consistency across Spring Social's supported bindings, we do recommend you consider these implementation technologies (and please let us know if they do not meet your needs).

Spring Social has adopted a convention where each API implementation class is named "{ProviderId}Template" e.g. TwitterTemplate. We favor this convention unless there is a good reason to deviate from it. As discussed in the previous section, we recommend keeping implementation types separate from the public API types. We also recommend keeping internal implementation details package-private.

The way in which an API binding implementation is constructed will vary based on the API's authorization protocol. For APIs secured with OAuth1, the consumerKey, consumerSecret, accessToken, and accessTokenSecret will be required for construction:

```
public TwitterTemplate(String consumerKey, String consumerSecret, String accessToken,
    String accessTokenSecret) { ... }
```

For OAuth2, only the access token should be required:

```
public FacebookTemplate(String accessToken) { ... }
```

Each request made to the API server needs to be signed with the authorization credentials provided during construction of the binding. This signing process consists of adding an "Authorization" header to each client request before it is executed. For OAuth1, the process is quite complicated, and is used to support an elaborate request signature verification algorithm between the client and server. For OAuth2, it is a lot simpler, but does still vary across the various drafts of the OAuth2 specification.

To encapsulate this complexity, for each authorization protocol Spring Social provides a ApiTemplate base class you may extend from to construct a pre-configured RestTemplate instance that performs the request signing for you. For OAuth1:

```
public class TwitterTemplate extends AbstractOAuth1ApiBinding {
    public TwitterTemplate(String consumerKey, String consumerSecret, String accessToken,
            String accessTokenSecret) {
        super(consumerKey, consumerSecret, accessToken, accessTokenSecret);
    }
}
```

An OAuth2 example:

```
public class FacebookTemplate extends AbstractOAuth2ApiBinding {
    public FacebookTemplate(String accessToken) {
        super(accessToken);
    }
}
```

Once configured as shown above, you simply implement call getRestTemplate() and implement the various API operations. The existing Spring Social client modules all invoke their RestTemplate instances in a standard manner:

```
public TwitterProfile getUserProfile() {
    return getRestTemplate().getForObject(buildUri("account/verify_credentials.json"),
        TwitterProfile.class);
}
```

A note on RestTemplate usage: we do favor the RestTemplate methods that accept a URI object instead of a uri String. This ensures we always properly encode client data submitted in URI query parameters, such as screen_name below:

```
public TwitterProfile getUserProfile(String screenName) {
    return getRestTemplate().getForObject(buildUri("users/show.json",
        Collections.singletonMap("screen_name", screenName)), TwitterProfile.class);
}
```

For complete implementation examples, consult the source of the existing API bindings included in Spring Social. The `spring-social-twitter` and `spring-social-facebook` modules provide particularly good references.

Testing a new Java API binding

As part of the spring-social-test module, Spring Social includes a framework for unit testing API bindings. This framework consists of a "MockRestServiceServer" that can be used to mock out API calls to the remote service provider. This allows for the development of independent, performant, automated unit tests that verify client API binding and object mapping behavior.

To use, first create a MockRestServiceServer against the RestTemplate instance used by your API implementation:

```
TwitterTemplate twitter = new TwitterTemplate("consumerKey", "consumerSecret", "accessToken",
    "accessTokenSecret");
MockRestServer mockServer = MockRestServiceServer.createServer(twitter.getRestTemplate());
```

Then, for each test case, record expectations about how the server should be invoked and answer what it should respond with:

```
@Test
public void getUserProfile() {
    HttpHeaders responseHeaders = new HttpHeaders();
    responseHeaders.setContentType(MediaType.APPLICATION_JSON);

    mockServer.expect(requestTo("https://api.twitter.com/1/account/verify_credentials.json"))
        .andExpect(method(GET))
        .andRespond(withResponse(jsonResource("verify-credentials"), responseHeaders));

    TwitterProfile profile = twitter.userOperations().getUserProfile();
    assertEquals(161064614, profile.getId());
    assertEquals("kdonald", profile.getScreenName());
}
```

In the example above the response body is written from a verify-credentials.json file located in the same package as the test class:

```
private Resource jsonResource(String filename) {
    return new ClassPathResource(filename + ".json", getClass());
}
```

The content of the file should mirror the content the remote service provider would return, allowing the client JSON deserialization behavior to be fully tested:

```
{
    "id":161064614,
    "screen_name":"kdonald"
}
```

For complete test examples, consult the source of the existing API bindings included in Spring Social. The `spring-social-twitter` and `spring-social-facebook` modules provide particularly good references.

Integrating an existing Java API binding

If you are adding support for a popular service provider, chances are a Java binding to the provider's API may already exist. For example, the Twitter4j library has been around for awhile and provides a complete binding to

Twitter's API. Instead of developing your own binding, you may simply wish to integrate what already exists. Spring Social's connect framework has been carefully designed to support this scenario.

To integrate an existing API binding, simply note the binding's primary API interface and implementation. For example, in Twitter4j the main API interface is named "Twitter" and instances are constructed by a TwitterFactory. You can always construct such an API instance directly, and you'll see in the following sections how to expose an instance as part of a Connection.

3.4 Creating a ServiceProvider model

As described in the previous section, a client binding to a secure API such as Facebook or Twitter requires valid user authorization credentials to work. Such credentials are generally obtained by having your application conduct an authorization "dance" or handshake with the service provider. Spring Social provides the ServiceProvider<A> abstraction to handle this "authorization dance". The abstraction also acts as a factory for native API (A) instances.

Since the authorization dance is protocol-specific, a ServiceProvider specialization exists for each authorization protocol. For example, if you are connecting to a OAuth2-based provider, you would implement OAuth2ServiceProvider. After you've done this, your implementation can be used to conduct the OAuth2 dance and obtain an authorized API instance. This is typically done in the context of a ConnectionFactory as part of establishing a new connection to the provider. The following sections describe the implementation steps for each ServiceProvider type.

OAuth2

To implement an OAuth2-based ServiceProvider, first create a subclass of AbstractOAuth2ServiceProvider named {ProviderId}ServiceProvider. Parameterize <A> to be the Java binding to the ServiceProvider's's API. Define a single constructor that accepts an clientId and clientSecret. Finally, implement getApi(String) to return a new API instance.

See `org.springframework.social.facebook.connect.FacebookServiceProvider` as an example OAuth2ServiceProvider:

```
public final class FacebookServiceProvider extends AbstractOAuth2ServiceProvider<Facebook> {

    public FacebookServiceProvider(String clientId, String clientSecret) {
        super(new OAuth2Template(clientId, clientSecret,
            "https://graph.facebook.com/oauth/authorize",
            "https://graph.facebook.com/oauth/access_token"));
    }

    public Facebook getApi(String accessToken) {
        return new FacebookTemplate(accessToken);
    }

}
```

In the constructor, you should call super, passing up the configured OAuth2Template that implements OAuth2Operations. The OAuth2Template will handle the "OAuth dance" with the provider, and should be

configured with the provided clientId and clientSecret, along with the provider-specific authorizeUrl and accessTokenUrl.

Some providers support provider sign-in (see Chapter 5, *Signing in with Service Provider Accounts*) through an authentication URL that is distinct from the authorization URL. Using the OAuth2Template constructor as shown above will assume that the authentication URL is the same as the authorization URL. But you may specify a different authentication URL by using OAuth2Template's other constructor. Facebook does not have a separate authentication URL, but for the sake of the example, suppose that Facebook's authentication URL is "https://graph.facebook.com/oauth/authenticate". The following implementation of the FacebookServiceProvider constructor configures the OAuth2Template for that case:

```
public FacebookServiceProvider(String clientId, String clientSecret) {
    super(new OAuth2Template(clientId, clientSecret,
        "https://graph.facebook.com/oauth/authorize",
        "https://graph.facebook.com/oauth/authenticate",
        "https://graph.facebook.com/oauth/access_token"));
}
```

In getApi(String), you should construct your API implementation, passing it the access token needed to make authorized requests for protected resources.

OAuth1

To implement an OAuth1-based ServiceProvider, first create a subclass of AbstractOAuth1ServiceProvider named {ProviderId}ServiceProvider. Parameterize <A> to be the Java binding to the ServiceProvider's API. Define a single constructor that accepts a consumerKey and consumerSecret. Finally, implement getApi(String, String) to return a new API instance.

See `org.springframework.social.twitter.connect.TwitterServiceProvider` as an example OAuth1ServiceProvider:

```
public final class TwitterServiceProvider extends AbstractOAuth1ServiceProvider<Twitter> {

    public TwitterServiceProvider(String consumerKey, String consumerSecret) {
        super(consumerKey, consumerSecret, new OAuth1Template(consumerKey, consumerSecret,
            "https://twitter.com/oauth/request_token",
            "https://twitter.com/oauth/authorize",
            "https://twitter.com/oauth/authenticate",
            "https://twitter.com/oauth/access_token"));
    }

    public Twitter getApi(String accessToken, String secret) {
        return new TwitterTemplate(getConsumerKey(), getConsumerSecret(), accessToken, secret);
    }

}
```

In the constructor, you should call super, passing up the the consumerKey, secret, and configured OAuth1Template. The OAuth1Template will handle the "OAuth dance" with the provider. It should be configured with the provided consumerKey and consumerSecret, along with the provider-specific

requestTokenUrl, authorizeUrl, authenticateUrl, and accessTokenUrl. The authenticateUrl parameter is optional and may be left out if the provider doesn't have an authentication URL that is different than the authorization URL.

As you can see here, OAuth1Template is constructed with Twitter's authentication URL (used for provider sign-in; see Chapter 5, *Signing in with Service Provider Accounts*), which is distinct from their authorization URL. Some providers don't have separate URLs for authentication and authorization. In those cases, you can use OAuth1Template's other constructor which doesn't take the authentication URL as a parameter. For example, here's how the TwitterServiceProvider constructor would look without configuring the authentication URL:

```
public TwitterServiceProvider(String consumerKey, String consumerSecret) {
    super(consumerKey, consumerSecret, new OAuth1Template(consumerKey, consumerSecret,
        "https://twitter.com/oauth/request_token",
        "https://twitter.com/oauth/authorize",
        "https://twitter.com/oauth/access_token"));
}
```

In getApi(String, String), you should construct your API implementation, passing it the four tokens needed to make authorized requests for protected resources.

Consult the JavaDoc API of the various service provider types for more information and subclassing options.

3.5 Creating an ApiAdapter

As discussed in the previous chapter, one of the roles of a Connection is to provide a common abstraction for a linked user account that is applied across all service providers. The role of the ApiAdapter is to map a provider's native API interface onto this uniform Connection model. A connection delegates to its adapter to perform operations such as testing the validity of its API credentials, setting metadata values, fetching a user profile, and updating user status:

```
public interface ApiAdapter<A> {

    boolean test(A api);

    void setConnectionValues(A api, ConnectionValues values);

    UserProfile fetchUserProfile(A api);

    void updateStatus(A api, String message);

}
```

Consider `org.springframework.social.twitter.connect.TwitterAdapter` as an example implementation:

```
public class TwitterAdapter implements ApiAdapter<Twitter> {

    public boolean test(Twitter twitter) {
        try {
```

```
            twitter.userOperations().getUserProfile();
            return true;
        } catch (ApiException e) {
            return false;
        }
    }

    public void setConnectionValues(Twitter twitter, ConnectionValues values) {
        TwitterProfile profile = twitter.userOperations().getUserProfile();
        values.setProviderUserId(Long.toString(profile.getId()));
        values.setDisplayName("@" + profile.getScreenName());
        values.setProfileUrl(profile.getProfileUrl());
        values.setImageUrl(profile.getProfileImageUrl());
    }

    public UserProfile fetchUserProfile(Twitter twitter) {
        TwitterProfile profile = twitter.userOperations().getUserProfile();
        return new UserProfileBuilder().setName(profile.getName()).setUsername(
            profile.getScreenName()).build();
    }

    public void updateStatus(Twitter twitter, String message) {
        twitter.timelineOperations().updateStatus(message);
    }

}
```

As you can see, test(...) returns true if the API instance is functional and false if it is not. setConnectionValues(...) sets the connection's providerUserId, displayName, profileUrl, and imageUrl properties from TwitterProfile data. fetchUserProfile(...) maps a TwitterProfile onto the normalized UserProfile model. updateStatus(...) update's the user's Twitter status. Consult the JavaDoc for ApiAdapter and Connection for more information and implementation guidance. We also recommend reviewing the other ApiAdapter implementations for additional examples.

3.6 Creating a ConnectionFactory

By now, you should have an API binding to the provider's API, a ServiceProvider<A> implementation for conducting the "authorization dance", and an ApiAdapter<A> implementation for mapping onto the uniform Connection model. The last step in adding support for a new service provider is to create a ConnectionFactory that wraps up these artifacts and provides a simple interface for establishing Connections. After this is done, you may use your connection factory directly, or you may add it to a registry where it can be used by the framework to establish connections in a dynamic, self-service manner.

Like a ServiceProvider<A>, a ConnectionFactory specialization exists for each authorization protocol. For example, if you are adding support for a OAuth2-based provider, you would extend from OAuth2ConnectionFactory. Implementation guidelines for each type are provided below.

OAuth2

Create a subclass of OAuth2ConnectionFactory<A> named {ProviderId}ConnectionFactory and parameterize A to be the Java binding to the service provider's API. Define a single constructor that accepts a clientId and clientSecret. Within the constructor call super, passing up the assigned providerId, a new

{ProviderId}ServiceProvider instance configured with the clientId/clientSecret, and a new {Provider}Adapter instance.

See `org.springframework.social.facebook.connect.FacebookConnectionFactory` as an example OAuth2ConnectionFactory:

```
public class FacebookConnectionFactory extends OAuth2ConnectionFactory<Facebook> {
    public FacebookConnectionFactory(String clientId, String clientSecret) {
        super("facebook", new FacebookServiceProvider(clientId, clientSecret), new FacebookAdapter());
    }
}
```

OAuth1

Create a subclass of OAuth1ConnectionFactory<A> named {ProviderId}ConnectionFactory and parameterize A to be the Java binding to the service provider's API. Define a single constructor that accepts a consumerKey and consumerSecret. Within the constructor call super, passing up the assigned providerId, a new {ProviderId}ServiceProvider instance configured with the consumerKey/consumerSecret, and a new {Provider}Adapter instance.

See `org.springframework.social.twitter.connect.TwitterConnectionFactory` as an example OAuth1ConnectionFactory:

```
public class TwitterConnectionFactory extends OAuth1ConnectionFactory<Facebook> {
    public TwitterConnectionFactory(String consumerKey, String consumerSecret) {
        super("twitter", new TwitterServiceProvider(consumerKey, consumerSecret), new TwitterAdapter());
    }
}
```

Consult the source and JavaDoc API for ConnectionFactory and its subclasses more information, examples, and advanced customization options.

4. Connecting to Service Providers

4.1 Introduction

In Chapter 2, *Service Provider 'Connect' Framework*, you learned how Spring Social's *Service Provider 'Connect' Framework* can be used to manage user connections that link your application's user accounts with accounts on external service providers. In this chapter, you'll learn how to control the connect flow in a web application environment.

Spring Social's `spring-social-web` module includes `ConnectController`, a Spring MVC controller that coordinates the connection flow between an application and service providers. `ConnectController` takes care of redirecting the user to the service provider for authorization and responding to the callback after authorization.

4.2 Configuring ConnectController

As `ConnectController` directs the overall connection flow, it depends on several other objects to do its job. Before getting into those, first we'll define a single Java @Configuration class where the various Spring Social objects, including ConnectController, will be configured:

```
@Configuration
public class SocialConfig {

}
```

Now, `ConnectController` first delegates to one or more ConnectionFactory instances to establish connections to providers on behalf of users. Once a connection has been established, it delegates to a `ConnectionRepository` to persist user connection data.

Each of the Spring Social provider modules includes a `ConnectionFactory` implementation:

- `org.springframework.social.twitter.connect.TwitterConnectionFactory`
- `org.springframework.social.facebook.connect.FacebookConnectionFactory`
- `org.springframework.social.linkedin.connect.LinkedInConnectionFactory`
- `org.springframework.social.tripit.connect.TripItConnectionFactory`
- `org.springframework.social.github.connect.GitHubConnectionFactory`
- `org.springframework.social.gowalla.connect.GowallaConnectionFactory`

To register one or more ConnectionFactories, simply define a ConnectionFactoryLocator @Bean as follows:

```
@Configuration
```

```
public class SocialConfig {

    @Bean
    public ConnectionFactoryLocator connectionFactoryLocator() {
        ConnectionFactoryRegistry registry = new ConnectionFactoryRegistry();

        registry.addConnectionFactory(new FacebookConnectionFactory(
            environment.getProperty("facebook.clientId"),
            environment.getProperty("facebook.clientSecret")));

        registry.addConnectionFactory(new TwitterConnectionFactory(
            environment.getProperty("twitter.consumerKey"),
            environment.getProperty("twitter.consumerSecret")));

        return registry;
    }

    @Inject
    private Environment environment;

}
```

Above, two connection factories, one for Facebook and one for Twitter, have been registered. If you would like to support other providers, simply register their connection factories here. Because client ids and secrets may be different across environments (e.g., test, production, etc), we recommend you externalize these values.

As discussed in Section 2.3, “Persisting connections”, `ConnectionRepository` defines operations for persisting and restoring connections for a specific user. Therefore, when configuring a `ConnectionRepository` bean for use by ConnectController, it must be scoped such that it can be created on a per-user basis. The following Java-based configuration shows how to construct an proxy to a request-scoped `ConnectionRepository` instance for the currently authenticated user:

```
@Configuration
public class SocialConfig {

    @Bean
    @Scope(value="request", proxyMode=ScopedProxyMode.INTERFACES)
    public ConnectionRepository connectionRepository(
        Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
        if (authentication == null) {
            throw new IllegalStateException("Unable to get a ConnectionRepository: no user signed in");
        }
        return usersConnectionRepository().createConnectionRepository(authentication.getName());
    }

}
```

The @Bean method above is injected with a `Principal` representing the current user's identity. This is passed to UsersConnectionRepository to construct a ConnectionRepository instance for that user.

This means that we're also going to need to configure a `UsersConnectionRepository` @Bean:

```
@Configuration
public class SocialConfig {
```

```
    @Bean
    public UsersConnectionRepository usersConnectionRepository() {
        return new JdbcUsersConnectionRepository(dataSource, connectionFactoryLocator(),
            textEncryptor);
    }

    @Inject
    private DataSource dataSource;

    @Inject
    private TextEncryptor textEncryptor;
}
```

UsersConnectionRepository is a singleton data store for connections across all users. `JdbcUsersConnectionRepository` is the RDMS-based implementation and needs a `DataSource`, `ConnectionFactoryLocator`, and `TextEncryptor` to do its job. It will use the `DataSource` to access the RDBMS when persisting and restoring connections. When restoring connections, it will use the `ConnectionFactoryLocator` to locate ConnectionFactory instances.

`JdbcUsersConnectionRepository` uses the `TextEncryptor` to encrypt credentials when persisting connections. Spring Security 3.1 makes a few useful text encryptors available via static factory methods in its `Encryptors` class. For example, a no-op text encryptor is useful at development time and can be configured like this:

```
@Configuration
public class SecurityConfig {

    @Configuration
    @Profile("dev")
    static class Dev {

        @Bean
        public TextEncryptor textEncryptor() {
            return Encryptors.noOpText();
        }

    }

}
```

Notice that the inner configuration class is annotated with `@Profile("dev")`. Spring 3.1 introduced the profile concept where certain beans will only be created when certain profiles are active. Here, the `@Profile` annotation ensures that this `TextEncryptor` will only be created when "dev" is an active profile. For production-time purposes, a stronger text encryptor is recommended and can be created when the "production" profile is active:

```
@Configuration
public class SecurityConfig {

    @Configuration
    @Profile("prod")
    static class Prod {
```

```
        @Bean
        public TextEncryptor textEncryptor() {
            return Encryptors.queryableText(environment.getProperty("security.encryptPassword"),
                environment.getProperty("security.encryptSalt"));
        }

        @Inject
        private Environment environment;

    }

}
```

Configuring connection support in XML

Up to this point, the connection support configuration has been done using Spring's Java-based configuration style. But you can configure it in either Java configuration or XML. Here's the XML equivalent of the `ConnectionFactoryRegistry` configuration:

```
<bean id="connectionFactoryLocator"
      class="org.springframework.social.connect.support.ConnectionFactoryRegistry">
    <property name="connectionFactories">
        <list>
            <bean class="org.springframework.social.twitter.connect.TwitterConnectionFactory">
                <constructor-arg value="${twitter.consumerKey}" />
                <constructor-arg value="${twitter.consumerSecret}" />
            </bean>
            <bean class="org.springframework.social.facebook.connect.FacebookConnectionFactory">
                <constructor-arg value="${facebook.clientId}" />
                <constructor-arg value="${facebook.clientSecret}" />
            </bean>
        </list>
    </property>
</bean>
```

This is functionally equivalent to the Java-based configuration of `ConnectionFactoryRegistry` shown before.

Here's an XML equivalent of the `JdbcUsersConnectionRepository` and `ConnectionRepository` configurations shown before:

```
<bean id="usersConnectionRepository"
      class="org.springframework.social.connect.jdbc.JdbcUsersConnectionRepository">
    <constructor-arg ref="dataSource" />
    <constructor-arg ref="connectionFactoryLocator" />
    <constructor-arg ref="textEncryptor" />
</bean>

<bean id="connectionRepository" factory-method="createConnectionRepository"
      factory-bean="usersConnectionRepository" scope="request">
    <constructor-arg value="#{request.userPrincipal.name}" />
    <aop:scoped-proxy proxy-target-class="false" />
</bean>
```

Likewise, here is the equivalent configuration of the `TextEncryptor` beans:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

    <beans profile="dev">
        <bean id="textEncryptor" class="org.springframework.security.crypto.encrypt.Encryptors"
            factory-method="noOpText" />
    </beans>

    <beans profile="prod">
        <bean id="textEncryptor" class="org.springframework.security.crypto.encrypt.Encryptors"
                factory-method="text">
            <constructor-arg value="${security.encryptPassword}" />
            <constructor-arg value="${security.encryptSalt}" />
        </bean>
    </beans>

</beans>
```

Just like the Java-based configuration, profiles are used to select which of the text encryptors will be created.

4.3 Creating connections with `ConnectController`

With its dependencies configured, `ConnectController` now has what it needs to allow users to establish connections with registered service providers. Now, simply add it to your Social @Configuration:

```
@Configuration
public class SocialConfig {

    @Bean
    public ConnectController connectController() {
        return new ConnectController(connectionFactoryLocator(),
            connectionRepository());
    }

}
```

Or, if you prefer Spring's XML-based configuration, then you can configure `ConnectController` like this:

```
<bean class="org.springframework.social.connect.web.ConnectController">
    <!-- relies on by-type autowiring for the constructor-args -->
</bean>
```

`ConnectController` supports authorization flows for OAuth 1 and OAuth 2, relying on `OAuth1Operations` or `OAuth2Operations` to handle the specifics for each protocol.

`ConnectController` will obtain the appropriate OAuth operations interface from one of the provider connection factories registered with `ConnectionFactoryRegistry`. It will select a specific `ConnectionFactory` to use by matching the connection factory's ID with the URL path. The path pattern that `ConnectController` handles is "/connect/{providerId}". Therefore, if `ConnectController` is handling a request for "/connect/twitter", then the `ConnectionFactory` whose `getProviderId()` returns "twitter" will be used. (As configured in the previous section, `TwitterConnectionFactory` will be chosen.)

When coordinating a connection with a service provider, `ConnectController` constructs a callback URL for the provider to redirect to after the user grants authorization. By default `ConnectController` uses information from the request to determine the protocol, host name, and port number to use when creating the callback URL. This is fine in many cases, but if your application is hosted behind a proxy those details may point to an internal server and will not be suitable for constructing a public callback URL.

If you have this problem, you can set the `applicationUrl` property to the base external URL of your application. `ConnectController` will use that URL to construct the callback URL instead of using information from the request. For example:

```
@Configuration
public class SocialConfig {

    @Bean
    public ConnectController connectController() {
        ConnectController controller = new ConnectController(
            connectionFactoryLocator(), connectionRepository());
        controller.setApplicationUrl(environment.getProperty("application.url");
        return controller;
    }

}
```

Or if you prefer XML configuration:

```
<bean class="org.springframework.social.connect.web.ConnectController">
    <!-- relies on by-type autowiring for the constructor-args -->
    <property name="applicationUrl" value="${application.url}" />
</bean>
```

Just as with the authorization keys and secrets, we recommend that you externalize the application URL because it will likely vary across different deployment environments.

The flow that `ConnectController` follows is slightly different, depending on which authorization protocol is supported by the service provider. For OAuth 2-based providers, the flow is as follows:

- `GET /connect` - Displays a web page showing connection status for all providers.
- `GET /connect/{providerId}` - Displays a web page showing connection status to the provider.
- `POST /connect/{providerId}` - Initiates the connection flow with the provider.

- `GET /connect/{providerId}?code={code}` - Receives the authorization callback from the provider, accepting an authorization code. Uses the code to request an access token and complete the connection.
- `DELETE /connect/{providerId}` - Severs all of the user's connection with the provider.
- `DELETE /connect/{providerId}/{providerUserId}` - Severs a specific connection with the provider, based on the user's provider user ID.

For an OAuth 1 provider, the flow is very similar, with only a subtle difference in how the callback is handled:

- `GET /connect` - Displays a web page showing connection status for all providers.
- `GET /connect/{providerId}` - Displays a web page showing connection status to the provider.
- `POST /connect/{providerId}` - Initiates the connection flow with the provider.
- `GET /connect/{providerId}?oauth_token={request token}&oauth_verifier={verifier}` - Receives the authorization callback from the provider, accepting a verification code. Exchanges this verification code along with the request token for an access token and completes the connection. The `oauth_verifier` parameter is optional and is only used for providers implementing OAuth 1.0a.
- `DELETE /connect/{providerId}` - Severs all of the user's connection with the provider.
- `DELETE /connect/{providerId}/{providerUserId}` - Severs a specific connection with the provider, based on the user's provider user ID.

Displaying a connection page

Before the connection flow starts in earnest, a web application may choose to show a page that offers the user information on their connection status. This page would offer them the opportunity to create a connection between their account and their social profile. `ConnectController` can display such a page if the browser navigates to `/connect/{provider}`.

For example, to display a connection status page for Twitter, where the provider name is "twitter", your application should provide a link similar to this:

```
<a href="<c:url value="/connect/twitter" />">Connect to Twitter</a>
```

`ConnectController` will respond to this request by first checking to see if a connection already exists between the user's account and Twitter. If not, then it will with a view that should offer the user an opportunity to create the connection. Otherwise, it will respond with a view to inform the user that a connection already exists.

The view names that `ConnectController` responds with are based on the provider's name. In this case, since the provider name is "twitter", the view names are "connect/twitterConnect" and "connect/twitterConnected".

Optionally, you may choose to display a page that shows connection status for all providers. In that case, the link might look like this:

```
<a href="<c:url value="/connect" />">Your connections</a>
```

The view name that `ConnectController` responds with for this URL is "connect/status".

Initiating the connection flow

To kick off the connection flow, the application should `POST` to `/connect/{providerId}`. Continuing with the Twitter example, a JSP view resolved from "connect/twitterConnect" might include the following form:

```
<form action="<c:url value="/connect/twitter" />" method="POST">
    <p>You haven't created any connections with Twitter yet. Click the button to create
       a connection between your account and your Twitter profile.
       (You'll be redirected to Twitter where you'll be asked to authorize the connection.)</p>
    <p><button type="submit"><img src="<c:url value="/resources/social/twitter/signin.png" />"/>
    </button></p>
</form>
```

When `ConnectController` handles the request, it will redirect the browser to the provider's authorization page. In the case of an OAuth 1 provider, it will first fetch a request token from the provider and pass it along as a parameter to the authorization page. Request tokens aren't used in OAuth 2, however, so instead it passes the application's client ID and redirect URI as parameters to the authorization page.

For example, Twitter's authorization URL has the following pattern:

```
https://twitter.com/oauth/authorize?oauth_token={token}
```

If the application's request token were "vPyVSe"[1], then the browser would be redirected to https://twitter.com/oauth/authorize?oauth_token=vPyVSe and a page similar to the following would be displayed to the user (from Twitter)[2]:

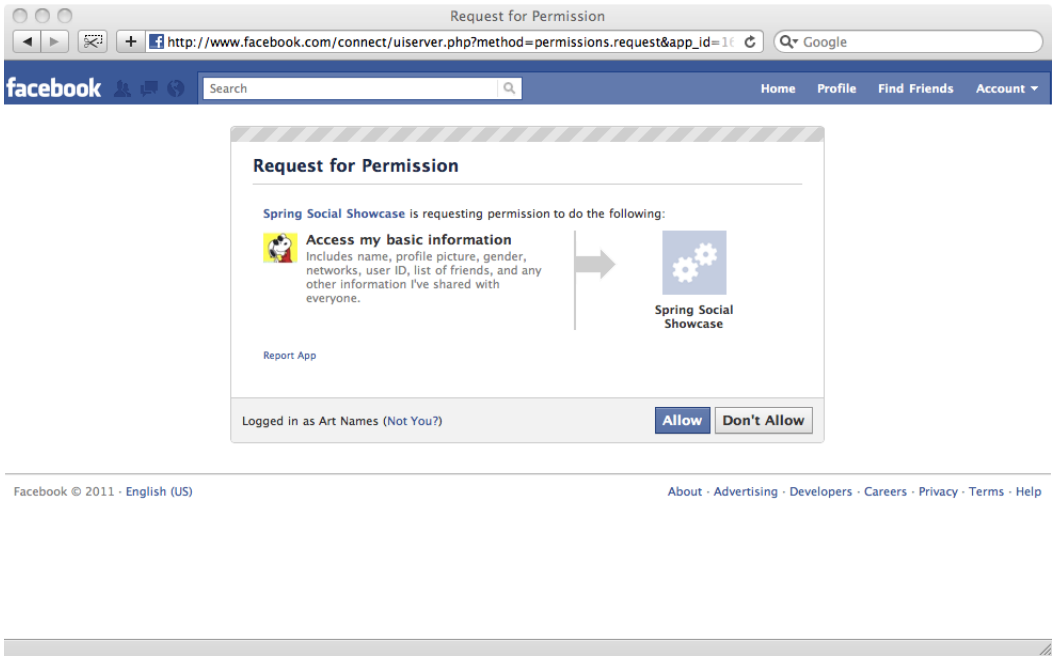
[1] This is just an example. Actual request tokens are typically much longer.

[2] If the user has not yet signed into Twitter, the authorization page will also include a username and password field for authentication into Twitter.

In contrast, Facebook is an OAuth 2 provider, so its authorization URL takes a slightly different pattern:

```
https://graph.facebook.com/oauth/authorize?client_id={clientId}&redirect_uri={redirectUri}
```

Thus, if the application's Facebook client ID is "0b754" and it's redirect URI is "http://www.mycoolapp.com/connect/facebook", then the browser would be redirected to https://graph.facebook.com/oauth/authorize?client_id=0b754&redirect_uri=http://www.mycoolapp.com/connect/facebook and Facebook would display the following authorization page to the user:



If the user clicks the "Allow" button to authorize access, the provider will redirect the browser back to the authorization callback URL where `ConnectController` will be waiting to complete the connection.

The behavior varies from provider to provider when the user denies the authorization. For instance, Twitter will simply show a page telling the user that they denied the application access and does not redirect back to the application's callback URL. Facebook, on the other hand, will redirect back to the callback URL with error information as request parameters.

Authorization scope

In the previous example of authorizing an application to interact with a user's Facebook profile, you notice that the application is only requesting access to the user's basic profile information. But there's much more that an application can do on behalf of a user with Facebook than simply harvest their profile data. For example, how can an application gain authorization to post to a user's Facebook wall?

OAuth 2 authorization may optionally include a scope parameter that indicates the type of authorization being requested. On the provider, the "scope" parameter should be passed along to the authorization URL. In the case of Facebook, that means that the Facebook authorization URL pattern should be as follows:

```
https://graph.facebook.com/oauth/authorize?client_id={clientId}&redirect_uri={redirectUri}&scope={scope}
```

`ConnectController` accepts a "scope" parameter at authorization and passes its value along to the provider's authorization URL. For example, to request permission to post to a user's Facebook wall, the connect form might look like this:

```
<form action="<c:url value="/connect/twitter" />" method="POST">
    <input type="hidden" name="scope" value="publish_stream,offline_access" />
    <p>You haven't created any connections with Twitter yet. Click the button to create
       a connection between your account and your Twitter profile.
       (You'll be redirected to Twitter where you'll be asked to authorize the connection.)</p>
    <p><button type="submit"><img src="<c:url value="/resources/social/twitter/signin.png" />"/>
    </button></p>
</form>
```

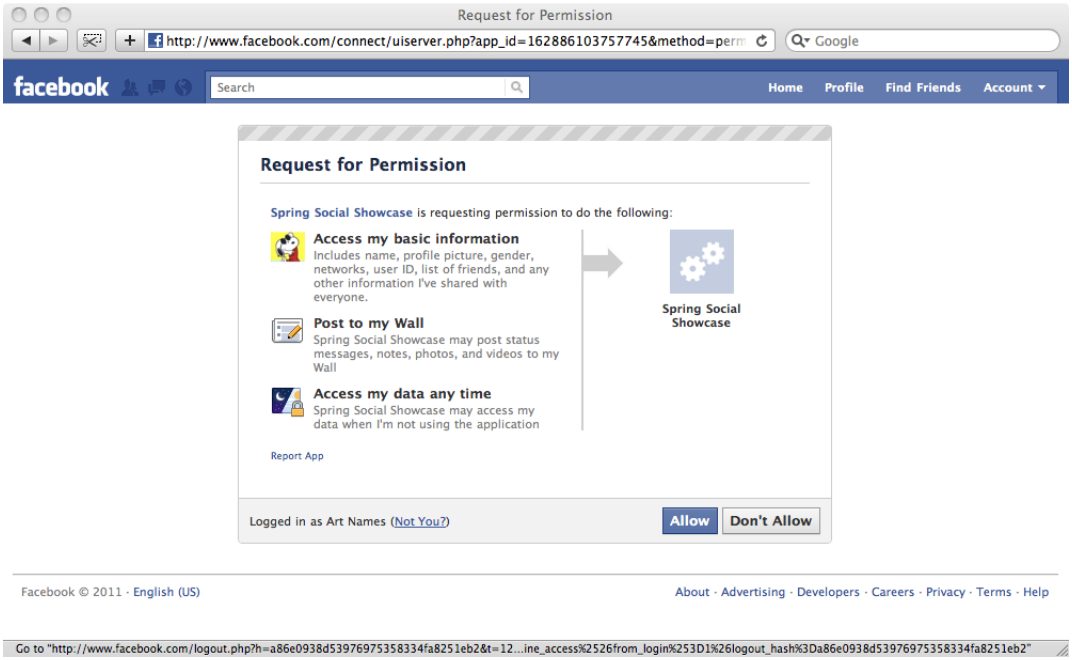
The hidden "scope" field contains the scope values to be passed along in the `scope>` parameter to Facebook's authorization URL. In this case, "publish_stream" requests permission to post to a user's wall. In addition, "offline_access" requests permission to access Facebook on behalf of a user even when the user isn't using the application.

Note

OAuth 2 access tokens typically expire after some period of time. Per the OAuth 2 specification, an application may continue accessing a provider after a token expires by using a refresh token to either renew an expired access token or receive a new access token (all without troubling the user to re-authorize the application).

Facebook does not currently support refresh tokens. Moreover, Facebook access tokens expire after about 2 hours. So, to avoid having to ask your users to re-authorize ever 2 hours, the best way to keep a long-lived access token is to request "offline_access".

When asking for "publish_stream,offline_access" authorization, the user will be prompted with the following authorization page from Facebook:



Scope values are provider-specific, so check with the service provider's documentation for the available scopes. Facebook scopes are documented at http://developers.facebook.com/docs/authentication/permissions.

Responding to the authorization callback

After the user agrees to allow the application have access to their profile on the provider, the provider will redirect their browser back to the application's authorization URL with a code that can be exchanged for an access token. For OAuth 1.0a providers, the callback URL is expected to receive the code (known as a verifier in OAuth 1 terms) in an `oauth_verifier` parameter. For OAuth 2, the code will be in a `code` parameter.

`ConnectController` will handle the callback request and trade in the verifier/code for an access token. Once the access token has been received, the OAuth dance is complete and the application may use the access token to interact with the provider on behalf of the user. The last thing that `ConnectController` does is to hand off the access token to the `ServiceProvider` implementation to be stored for future use.

Disconnecting

To delete a connection via `ConnectController`, submit a DELETE request to "/connect/{provider}".

In order to support this through a form in a web browser, you'll need to have Spring's `HiddenHttpMethodFilter` [http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/web/filter/HiddenHttpMethodFilter.html] configured in your application's web.xml. Then you can provide a disconnect button via a form like this:

```
<form action="<c:url value="/connect/twitter" />" method="post">
  <div class="formInfo">
    <p>
      Spring Social Showcase is connected to your Twitter account.
      Click the button if you wish to disconnect.
    </p>
  </div>
```

```
  <button type="submit">Disconnect</button>
  <input type="hidden" name="_method" value="delete" />
</form>
```

When this form is submitted, `ConnectController` will disconnect the user's account from the provider. It does this by calling the `disconnect()` method on each of the `Connection`s returned by the provider's `getConnections()` method.

4.4 Connection interceptors

In the course of creating a connection with a service provider, you may want to inject additional functionality into the connection flow. For instance, perhaps you'd like to automatically post a tweet to a user's Twitter timeline immediately upon creating the connection.

`ConnectController` may be configured with one or more connection interceptors that it will call at points in the connection flow. These interceptors are defined by the `ConnectInterceptor` interface:

```
public interface ConnectInterceptor<A> {

    void preConnect(ConnectionFactory<A> connectionFactory, MultiValueMap<String, String> parameters, WebReques

    void postConnect(Connection<A> connection, WebRequest request);

}
```

The `preConnect()` method will be called by `ConnectController` just before redirecting the browser to the provider's authorization page. Custom authorization parameters may be added to the provided parameter map. `postConnect()` will be called immediately after a connection has been persisted linking the user's local account with the provider profile.

For example, suppose that after connecting a user account with their Twitter profile you want to immediately post a tweet about that connection to the user's Twitter timeline. To accomplish that, you might write the following connection interceptor:

```
public class TweetAfterConnectInterceptor implements ConnectInterceptor<Twitter> {

    public void preConnect(ConnectionFactory<TwitterApi> provider, MultiValueMap<String, String> parameters, We
        // nothing to do
    }

    public void postConnect(Connection<TwitterApi> connection, WebRequest request) {
        connection.updateStatus("I've connected with the Spring Social Showcase!");
    }
}
```

This interceptor can then be injected into `ConnectController` when it is created:

```
@Bean
```

```
public ConnectController connectController() {
    ConnectController controller = new ConnectController(connectionFactoryLocator(),
        connectionRepository());
    controller.addInterceptor(new TweetAfterConnectInterceptor());
    return controller;
}
```

Or, as configured in XML:

```
<bean class="org.springframework.social.connect.web.ConnectController">
    <property name="interceptors">
        <list>
            <bean class="org.springframework.social.showcase.twitter.TweetAfterConnectInterceptor" />
        </list>
    </property>
</bean>
```

Note that the `interceptors` property is a list and can take as many interceptors as you'd like to wire into it. When it comes time for `ConnectController` to call into the interceptors, it will only invoke the interceptor methods for those interceptors whose service operations type matches the service provider's operations type. In the example given here, only connections made through a service provider whose operation type is `TwitterApi` will trigger the interceptor's methods.

5. Signing in with Service Provider Accounts

5.1 Introduction

In order to ease sign in for their users, many applications allow sign in with a service provider such as Twitter or Facebook. With this authentication technique, the user signs into (or may already be signed into) his or her provider account. The application then tries to match that provider account to a local user account. If a match is found, the user is automatically signed into the application.

Spring Social supports such service provider-based authentication with `ProviderSignInController` from the `spring-social-web` module. `ProviderSignInController` works very much like `ConnectController` in that it goes through the OAuth flow (either OAuth 1 or OAuth 2, depending on the provider). Instead of creating a connection at the end of process, however, `ProviderSignInController` attempts to find a previously established connection and uses the connected account to authenticate the user with the application. If no previous connection matches, the flow will be sent to the application's sign up page so that the user may register with the application.

5.2 Enabling provider sign in

To add provider sign in capability to your Spring application, configure `ProviderSignInController` as a bean in your Spring MVC application:

```
@Bean
public ProviderSignInController providerSignInController() {
    return new ProviderSignInController(connectionFactoryLocator(),
        usersConnectionRepository(), new SimpleSignInAdapter());
}
```

Or in XML, if you prefer:

```
<bean class="org.springframework.social.connect.signin.web.ProviderSignInController">
    <!-- relies on by-type autowiring for the constructor-args -->
</bean>
```

As with `ConnectController`, `ProviderSignInController` uses information from the request to determine the protocol, host name, and port number to use when creating a callback URL. But you may set the `applicationUrl` property to the base external URL of your application to overcome any problems where the request refers to an internal server. For example:

```
@Bean
public ProviderSignInController providerSignInController() {
    ProviderSignInController controller = new ProviderSignInController(connectionFactoryLocator(),
        usersConnectionRepository(), new SimpleSignInAdapter());
    controller.setApplicationUrl(environment.getProperty("application.url"));
    return controller;
}
```

Or when configured in XML:

```
<bean class="org.springframework.social.connect.signin.web.ProviderSignInController">
    <!-- relies on by-type autowiring for the constructor-args -->
    <property name="applicationUrl" value="${application.url}" />
</bean>
```

Once again, we recommend that you externalize the value of the application URL since it will vary between deployment environments.

When authenticating via an OAuth 2 provider, `ProviderSignInController` supports the following flow:

- `POST /signin/{providerId}` - Initiates the sign in flow by redirecting to the provider's authentication endpoint.
- `GET /signin/{providerId}?code={verifier}` - Receives the authentication callback from the provider, accepting a code. Exchanges this code for an access token. Using this access token, it retrieves the user's provider user ID and uses that to lookup a connected account and then authenticates to the application through the sign in service.
  - If the provider user ID doesn't match any existing connection, `ProviderSignInController` will redirect to a sign up URL. The default sign up URL is "/signup" (relative to the application root), but can be customized by setting the `signUpUrl` property.
  - If the provider user ID matches more than one existing connection, `ProviderSignInController` will redirect to the application's sign in URL to offer the user a chance to sign in through another provider or with their username and password. The request to the sign in URL will have an "error" query parameter set to "multiple_users" to indicate the problem so that the page can communicate it to the user. The default sign in URL is "/signin" (relative to the application root), but can be customized by setting the `signInUrl` property.

For OAuth 1 providers, the flow is only slightly different:

- `POST /signin/{providerId}` - Initiates the sign in flow. This involves fetching a request token from the provider and then redirecting to Provider's authentication endpoint.
- `GET /signin/{providerId}?oauth_token={request token}&oauth_verifier={verifier}` - Receives the authentication callback from the provider, accepting a verification code. Exchanges this verification code along with the request token for an access token. Using this access token, it retrieves the user's provider user ID and uses that to lookup a connected account and then authenticates to the application through the sign in service.
  - If the provider user ID doesn't match any existing connection, `ProviderSignInController` will redirect to a sign up URL. The default sign up URL is "/signup" (relative to the application root), but can be customized by setting the `signUpUrl` property.

- If the provider user ID matches more than one existing connection, `ProviderSignInController` will redirect to the application's sign in URL to offer the user a chance to sign in through another provider or with their username and password. The request to the sign in URL will have an "error" query parameter set to "multiple_users" to indicate the problem so that the page can communicate it to the user. The default sign in URL is "/signin" (relative to the application root), but can be customized by setting the `signInUrl` property.

ProviderSignInController's dependencies

As shown in the Java-based configuration above, `ProviderSignInController` depends on a handful of other objects to do its job.

- A `ConnectionFactoryLocator` to lookup the ConnectionFactory used to create the Connection to the provider.
- A `UsersConnectionRepository` to find the user that has the connection to the provider user attempting to sign in.
- A `SignInAdapter` to sign a user into the application when a matching connection is found.

When using XML configuration, it isn't necessary to explicitly configure these constructor arguments because `ProviderSignInController`'s constructor is annotated with `@Inject`. Those dependencies will be given to `ProviderSignInController` via autowiring. You'll still need to make sure they're available as beans in the Spring application context so that they can be autowired.

You should have already configured most of these dependencies when setting up connection support (in the previous chapter). But when used with `ProviderSignInController`, you should configure them to be created as scoped proxies:

```
@Bean
@Scope(value="singleton", proxyMode=ScopedProxyMode.INTERFACES)
public ConnectionFactoryLocator connectionFactoryLocator() {
    ConnectionFactoryRegistry registry = new ConnectionFactoryRegistry();

    registry.addConnectionFactory(new FacebookConnectionFactory(
        environment.getProperty("facebook.clientId"),
        environment.getProperty("facebook.clientSecret")));

    registry.addConnectionFactory(new TwitterConnectionFactory(
        environment.getProperty("twitter.consumerKey"),
        environment.getProperty("twitter.consumerSecret")));

    return registry;
}

@Bean
@Scope(value="singleton", proxyMode=ScopedProxyMode.INTERFACES)
public UsersConnectionRepository usersConnectionRepository() {
    return new JdbcUsersConnectionRepository(dataSource, connectionFactoryLocator(), textEncryptor);
}
```

In the event that the sign in attempt fails, the sign in attempt will be stored in the session to be used to present a sign-up page to the user (see Section 5.3, “Signing up after a failed sign in”). By configuring ConnectionFactoryLocator and UsersConnectionRepository as scoped proxies, it enables the proxies to be carried along with the sign in attempt in the session rather than the actual objects themselves.

The `SignInAdapter` is exclusively used for provider sign in and so a `SignInAdapter` bean will need to be added to the configuration. But first, you'll need to write an implementation of the `SignInAdapter` interface.

The `SignInAdapter` interface is defined as follows:

```
public interface SignInAdapter {
    void signIn(String userId, Connection<?> connection, NativeWebRequest request);
}
```

The `signIn()` method takes the local application user's user ID normalized as a `String`. No other credentials are necessary here because by the time this method is called the user will have signed into the provider and their connection with that provider has been used to prove the user's identity. Implementations of this interface should use this user ID to authenticate the user to the application.

Different applications will implement security differently, so each application must implement `SignInAdapter` in a way that fits its unique security scheme. For example, suppose that an application's security is based on Spring Security and simply uses a user's account ID as their principal. In that case, a simple implementation of `SignInAdapter` might look like this:

```
@Service
public class SpringSecuritySignInAdapter implements SignInAdapter {
    public void signIn(String localUserId, Connection<?> connection, NativeWebRequest request) {
        SecurityContextHolder.getContext().setAuthentication(
            new UsernamePasswordAuthenticationToken(localUserId, null, null));
    }
}
```

Adding a provider sign in button

With `ProviderSignInController` and a `SignInAdapter` configured, the backend support for provider sign in is in place. The last thing to do is to add a sign in button to your application that will kick off the authentication flow with `ProviderSignInController`.

For example, the following HTML snippet adds a "Signin with Twitter" button to a page:

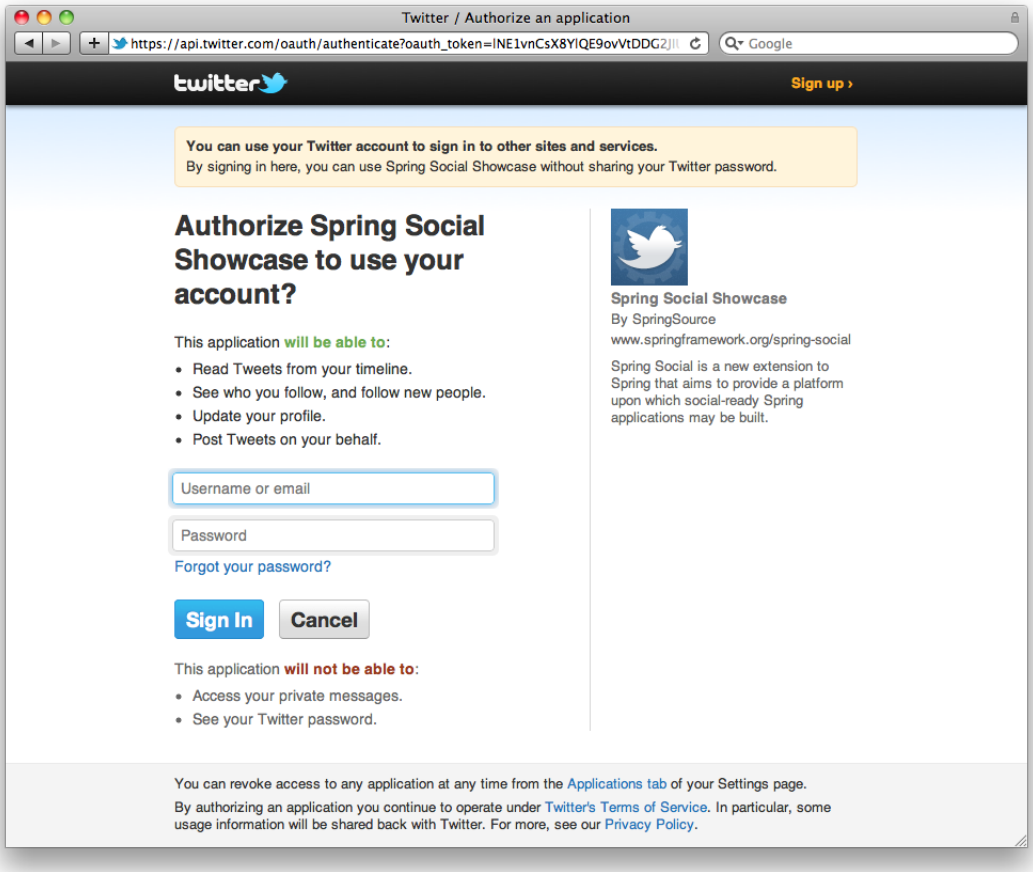
```
<form id="tw_signin" action="<c:url value="/signin/twitter"/>" method="POST">
  <button type="submit">
    <img src="<c:url value="/resources/social/twitter/sign-in-with-twitter-d.png"/>" />
  </button>
</form>
```

Notice that the path used in the form's `action` attribute maps to the first step in `ProviderSignInController`'s flow. In this case, the provider is identified as "twitter".

Note

Some providers offer client-side sign in widgets, such as Twitter @Anywhere's "Connect with Twitter" button and Facebook's `<fb:login-button>`. Although these widgets offer a sign in experience similar to that of `ProviderSignInController`, they cannot be used to drive `ProviderSignInController`'s sign in flow. The `ProviderSignInController` sign in flow should be initiated by submitting a POST request as described above.

Clicking this button will trigger a POST request to "/signin/twitter", kicking off the Twitter sign in flow. If the user has not yet signed into Twitter, the user will be presented with the following page from Twitter:



After signing in, the flow will redirect back to the application to complete the sign in process.

5.3 Signing up after a failed sign in

If `ProviderSignInController` can't find a local user associated with a provider user attempting to sign in, there may be an opportunity to have the user sign up with the application. Leveraging the information about the user received from the provider, the user may be presented with a pre-filled sign up form to explicitly sign up with the application. It's also possible to use the user's provider data to implicitly create a new local application user without presenting a sign up form.

Signing up with a sign up form

By default, the sign up URL is "/signup", relative to the application root. You can override that default by setting the `signUpUrl` property on the controller. For example, the following configuration of `ProviderSignInController` sets the sign up URL to "/register":

```
@Bean
public ProviderSignInController providerSignInController() {
    ProviderSignInController controller = new ProviderSignInController(connectionFactoryLocator(),
        usersConnectionRepository(), new SimpleSignInAdapter());
    controller.setSignUpUrl("/register");
    return controller;
}
```

Or to set the sign up URL using XML configuration:

```
<bean class="org.springframework.social.connect.signin.web.ProviderSignInController">
    <property name="signUpUrl" value="/register" />
</bean>
```

Before redirecting to the sign up page, `ProviderSignInController` collects some information about the authentication attempt. This information can be used to prepopulate the sign up form and then, after successful sign up, to establish a connection between the new account and the provider account.

To prepopulate the sign up form, you can fetch the user profile data from a connection retrieved from `ProviderSignInUtils.getConnection()`. For example, consider this Spring MVC controller method that setups up the sign up form with a `SignupForm` to bind to the sign up form:

```
@RequestMapping(value="/signup", method=RequestMethod.GET)
public SignupForm signupForm(WebRequest request) {
    Connection<?> connection = ProviderSignInUtils.getConnection(request);
    if (connection != null) {
        return SignupForm.fromProviderUser(connection.fetchUserProfile());
    } else {
        return new SignupForm();
    }
}
```

If `ProviderSignInUtils.getConnection()` returns a connection, that means there was a failed provider sign in attempt that can be completed if the user registers to the application. In that case, a `SignupForm` object is created from the user profile data obtained from the connection's `fetchUserProfile()` method. Within `fromProviderUser()`, the `SignupForm` properties may be set like this:

```
public static SignupForm fromProviderUser(UserProfile providerUser) {
    SignupForm form = new SignupForm();
    form.setFirstName(providerUser.getFirstName());
    form.setLastName(providerUser.getLastName());
```

```
    form.setUsername(providerUser.getUsername());
    form.setEmail(providerUser.getEmail());
    return form;
}
```

Here, the `SignupForm` is created with the user's first name, last name, username, and email from the `UserProfile`. In addition, `UserProfile` also has a `getName()` method which will return the user's full name as given by the provider.

The availability of `UserProfile`'s properties will depend on the provider. Twitter, for example, does not provide a user's email address, so the `getEmail()` method will always return null after a sign in attempt with Twitter.

After the user has successfully signed up in your application a connection can be created between the new local user account and their provider account. To complete the connection call `ProviderSignInUtils.handlePostSignUp()`. For example, the following method handles the sign up form submission, creates an account and then calls `ProviderSignInUtils.handlePostSignUp()` to complete the connection:

```
@RequestMapping(value="/signup", method=RequestMethod.POST)
public String signup(@Valid SignupForm form, BindingResult formBinding, WebRequest request) {
    if (formBinding.hasErrors()) {
        return null;
    }
    Account account = createAccount(form, formBinding);
    if (account != null) {
        SignInUtils.signin(account.getUsername());
        ProviderSignInUtils.handlePostSignUp(account.getUsername(), request);
        return "redirect:/";
    }
    return null;
}
```

Implicit sign up

To enable implicit sign up, you must create an implementation of the `ConnectionSignUp` interface and inject an instance of that `ConnectionSignUp` to the connection repository. The `ConnectionSignUp` interface is simple, with only a single method to implement:

```
public interface ConnectionSignUp {
    String execute(Connection<?> connection);
}
```

The `execute()` method is given a `Connection` that it can use to retrieve information about the user. It can then use that information to create a new local application user and return the new local user ID. For example, the following implementation fetches the user's provider profile and uses it to create a new account:

```
public class AccountConnectionSignUp implements ConnectionSignUp {
```

```
    private final AccountRepository accountRepository;

    public AccountConnectionSignUp(AccountRepository accountRepository) {
        this.accountRepository = accountRepository;
    }

    public String execute(Connection<?> connection) {
        UserProfile profile = connection.fetchUserProfile();
        Account account = createAccount(profile);
        return account != null ? account.getUsername() : null;
    }

    private Account createAccount(UserProfile profile) {
        if (profile == null || profile.getUsername() == null) {
            return null;
        }
        Account account = new Account(profile.getUsername(), profile.getFirstName(), profile.getLastName());
        accountRepository.createAccount(account);
        return account;
    }

}
```

If `execute()` returns null, then it indicates that the user could not be implicitly signed up. In that case, `ProviderSignInController`'s explicit sign up flow will be in effect and the browser will be redirected to the sign up form.

Once you've written a `ConnectionSignUp` for your application, you'll need to inject it into the `UsersConnectionRepository`. In Java-based configuration:

```
@Bean
@Scope(value="singleton", proxyMode=ScopedProxyMode.INTERFACES)
public UsersConnectionRepository usersConnectionRepository(AccountRepository accountRepository) {
    JdbcUsersConnectionRepository repository = new JdbcUsersConnectionRepository(
        dataSource, connectionFactoryLocator(), Encryptors.noOpText());
    repository.setConnectionSignUp(new AccountConnectionSignUp(accountRepository));
    return repository;
}
```